

4

PARALLEL PROGRAMMING WITH MPI

In this chapter we describe the parallel programming scheme used to implement our code with message passing interface (MPI). In the first section we provide a brief overview of the architectures that support programs running in parallel. In the second section we give a brief introduction to MPI. The third section demonstrates the frequently used MPI subroutines by analysing a sample of the parallelized program for Poiseuille flow which we have developed and implemented.

4.1 Parallel architectures

The main components in a typical computer system are the processor, memory, input/output devices, and the communication channels that connect them. The processor is the workhorse of the system; it is the component that executes a program by performing arithmetic and logical operations on data. It is the only component that creates new information by combining or modifying current information. In a typical system there will be only one processor, known as the *central processing unit*, or *CPU*. Modern high performance systems, for example vector processors and parallel processors, often have more than one processor. Systems with only one processor are *serial processors*, or, especially among computational scientists, *scalar processors*.

Memory is a passive component that simply stores information until it is requested by another part of the system. During normal operations it feeds instructions and data

to the processor, and at other times it is the source or destination of data transferred by I/O devices. Information in a memory is accessed by its *address*.

Input/output (I/O) devices transfer information without altering it between the external world and one or more internal components. I/O devices can be secondary memories, for example disks and tapes, or devices used to communicate directly with users, such as video displays, keyboards, and mice.

The communication channels that tie the system together can either be simple links that connect two devices or more complex *switches* that interconnect several components and allow any two of them to communicate at a given point in time. When a switch is configured to allow two devices to exchange information, all other devices that rely on the switch are *blocked*, i.e. they must wait until the switch can be reconfigured.

The architecture of parallel computers can be categorized in terms of two aspects: whether the memory is physically centralized or distributed, and whether or not the address space is shared. A shared memory architecture uses shared system resources such as memory and I/O subsystem that can be accessed equally from all the processors shown in Fig. 4.1. The processors communicate with one another by one processor writing data into a location in memory and another processor reading the data. The advantage of this type of architecture is that it is easy to program as there are no explicit communications between processors as communications are handled via the global memory store. However, a bottleneck happens when a number of processors attempt to access the global memory store at the same time.

The distributed memory architecture illustrated in Fig. 4.2 gives each processor its own memory. A processor can only access the memory which is attached directly to it. If a processor needs data which is contained in the memory of a remote processor, then it must send a message to the remote processor asking it to send the data. In spite of the drawback of explicit communications in the distributed memory architecture, it is

much faster to access the local memory and far more scalable than the shared memory architecture.

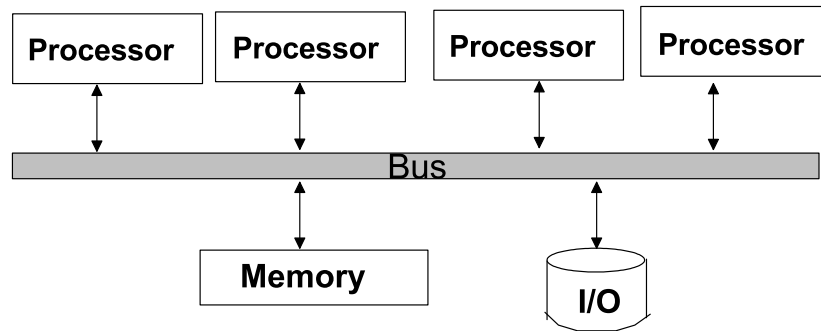


Figure 4.1: Shared memory architecture.

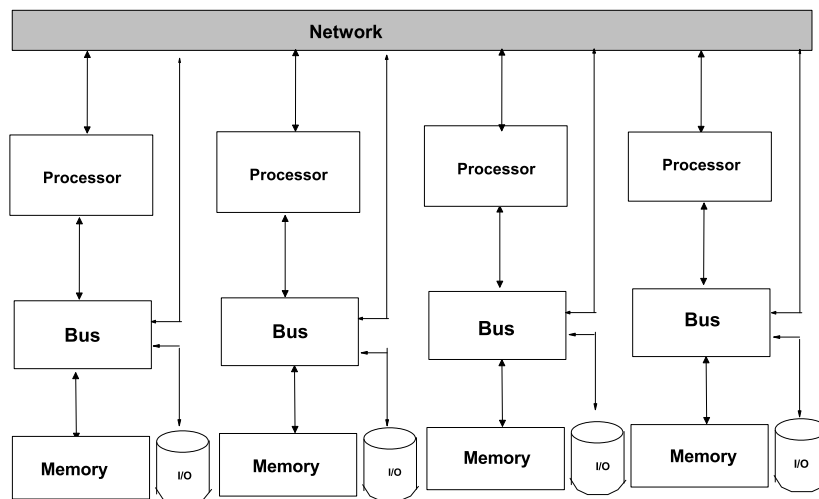


Figure 4.2: Distributed memory architecture.

4.2 Introduction to MPI

MPI is a message passing interface used for parallel processing in distributed memory systems [61]. MPI is a library of routines that can be called from C, C++, FORTRAN77 and FORTRAN90 programs. A single user program is prepared, but is run on multiple processes. Each instance of the program is assigned a unique process identifier, so that it is labelled and known which process it is. This allows the same program to be executed, but different tasks are performed in each process. By convention, the user sets up one process as a master, and the others as workers, but this is not necessary. Each process has its own set of data, and can communicate directly with other processes by passing data around. Because the data is distributed, it is likely that a computation on one process will require that a data value be copied from another process. For example, if process *A* needs the value of data item *X* that is stored in the memory of process *B*, then the program must include the following lines.

```
    if ( I am processor A ) then
    call MPI_Send ( X )
    else if ( I am processor B ) then
    call MPI_Recv ( X )
    end
```

We keep the lines as simple as possible to illustrate a number of important features of message passing interface.

4.2.1 Frequently used MPI subroutines

Frequently used environmental management and collective communication subroutines are introduced here.

4.2.1.1 *Environmental management subroutines*

Four environmental management subroutines, MPI_INIT, MPI_COMM_SIZE, MPI_COMM_RANK and MPI_FINALIZE are described.

CALL MPI_INIT(*ierror*)

Purpose Initializes MPI.

Parameters INTEGER *ierror* is the Fortran return code.

Description This routine initializes MPI. All MPI programs must call this routine once and only once before any other MPI subroutine. In Fortran, the return code of every MPI subroutine is given in the last argument of its subroutine call. If an MPI subroutine call is done successfully, the return code is 0; otherwise, a non zero value is returned.

CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierror)

Purpose Returns the number of processes belonging to the communicator specified in the first argument. A communicator is an identifier associated with a group of processes.

Parameters INTEGER *MPI_COMM_WORLD* is the communicator

 INTEGER *nprocs* is an integer used to specify the number of processes in the group.

 INTEGER *ierror* is the Fortran return code.

Description This routine returns the size of the group associated with a communicator, *MPI_COMM_WORLD*, which is a MPI related parameter and defined in the header file *mpif.h*. All Fortran procedures that use MPI subroutines have to include this file.

CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

Purpose Returns the rank of the local process in the group associated with a communicator.

Parameters INTEGER *MPI_COMM_WORLD* is the communicator.

 INTEGER *rank* is an integer used to specify the calling processes in the group.

 INTEGER *ierror* is the FORTRAN return code.

Description This routine returns the rank of the local process in the group associated with a communicator. Each process in a communicator has its unique rank, which is in the range from 0 to (*SIZE* - 1), where *SIZE* is the number of processes in that communicator and it is the return value of *MPI_COMM_SIZE*.

CALL MPI_FINALIZE(ierr)

Purpose Terminates all MPI processing.

Parameters INTEGER *ierror* is the Fortran return code.

Description This routine is the last MPI call. Any MPI calls made after MPI_FINALIZE raise an error. All pending communications involving a process have to be completed and all files opened by the process have to be closed before the process calls MPI_FINALIZE. Although MPI_FINALIZE terminates MPI processing, it does not terminate the process. It is possible to continue with non-MPI processing after calling MPI_FINALIZE, but no other MPI calls (including MPI_INIT) can be made.

4.2.1.2 *Collective communication subroutines*

A group of processes can exchange data by collective communication. The communicator argument in the collective communication subroutine specifies which processes are involved in the communication. In other words, all the processes belonging to that communicator must call the same collective communication subroutine with matching arguments. The commonly used patterns of collective communication are illustrated in Fig. 4.3 and discussed in the pages that follow.

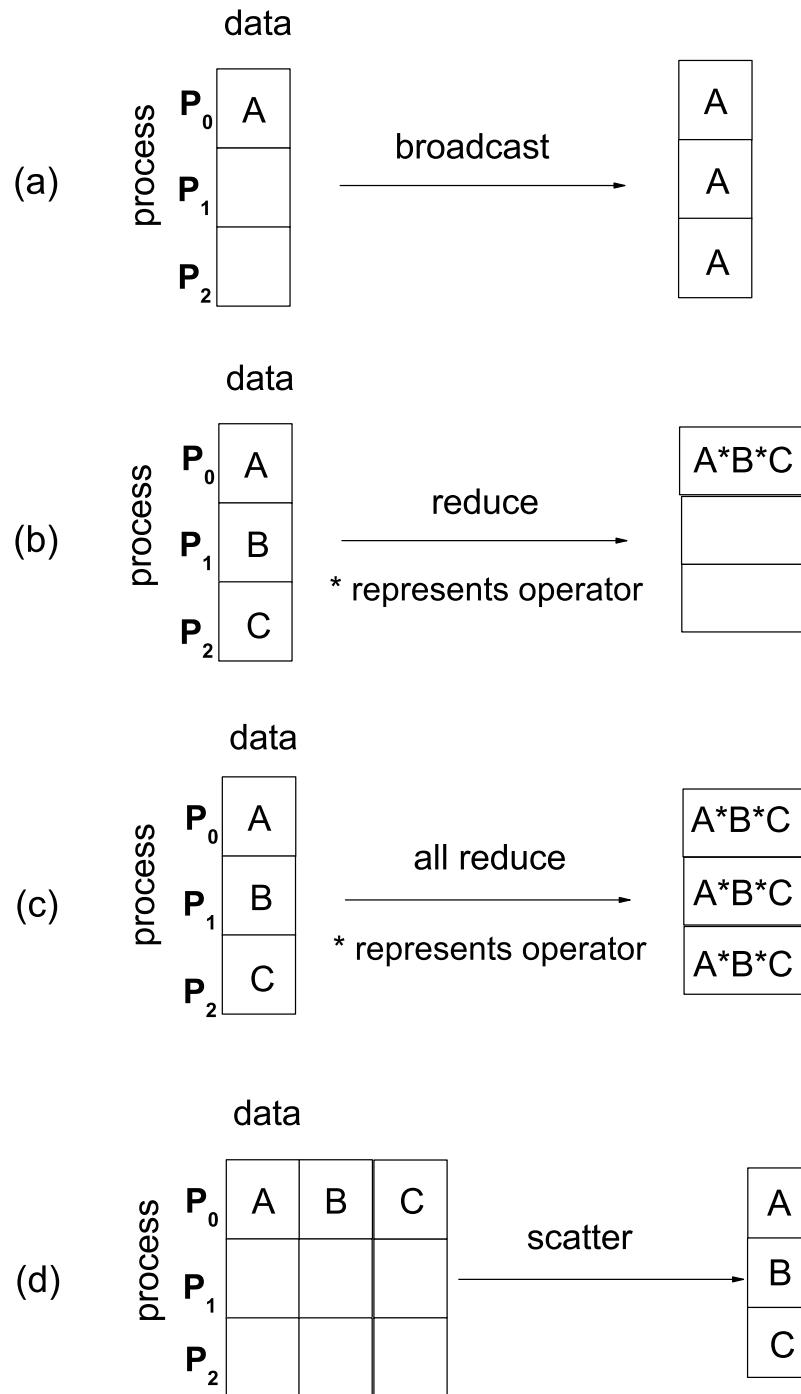


Figure 4.3: Patterns of collective communication. (a) Message broadcast (see `MPI_BCAST`); (b) and (c) reduction operations (see `MPI_REDUCE`); (d) distribution of individual messages (see `MPI_SCATTER`).

CALL MPI_BCAST(buffer, count, datatype, root, comm, ierror)

Purpose Broadcasts a message from root to all processes in communicator, *comm*.

Parameters *buffer*, the starting address of the buffer.
INTEGER *count*, the number of elements in the buffer.
INTEGER *datatype*, the data type of the buffer elements.
INTEGER *root*, the rank of the root process.
INTEGER *comm*, the communicator.
INTEGER *ierror*, the Fortran return code.

Description This routine broadcasts a message from root to all processes in *comm*. The contents of root buffer is copied to all processes. The amount of data sent must be equal to the amount of data received. All processes in *comm* need to call this routine.

CALL MPI_SCATTER(*sendbuf*, *sendcount*, *sendtype*, *recvbuf*,
recvcount, *recvtype*, *root*, *comm*, *ierror*)

Purpose Distributes individual messages from root to each process in *comm*.

Parameters *sendbuf*, the starting address of the send buffer.
INTEGER *sendcount*, the number of elements to be sent to each process, not the number of total elements to be sent from root.
INTEGER *sendtype*, the data type of the send buffer elements.
recvbuf, the starting address of the receive buffer.
INTEGER *recvcount*, the number of elements in the receive buffer.
INTEGER *recvtype*, the data type of the receive buffer elements.
INTEGER *root*, the rank of the sending process.
INTEGER *comm*, the communicator.
INTEGER *ierror*, the Fortran return code.

Description This routine distributes individual messages from root to each process in *comm*. The number of elements sent to each process is the same (*sendcount*). The first *sendcount* elements are sent to process 0, the next *sendcount* elements are sent to process 1, and so on. This routine is the inverse operation to MPI_GATHER. The amount of data sent must be equal to the amount of data received. All processes in *comm* need to call this routine.

CALL MPIREDUCE(*sendbuf*, *recvbuf*, *count*, *datatype*, *op*,
 root, *comm*, *ierror*)

Purpose Applies a reduction operation to the vector *sendbuf* over
the set of processes specified by *comm* and places the
result in *recvbuf* on root.

Parameters *sendbuf*, the starting address of the send buffer.
recvbuf, the starting address of the receive buffer.
INTEGER *count*, the number of elements in the send
buffer.
INTEGER *datatype* The data type of elements of the
send buffer.
INTEGER *op*, the reduction operation.
INTEGER *root*, the rank of the root process.
INTEGER *comm*, the communicator.
INTEGER *ierror*, the Fortran return code.

Description This routine applies a reduction operation to the vec-
tor *sendbuf* over the set of processes specified by *comm*
and places the result in *recvbuf* on root. Both the in-
put and output buffers have the same number of el-
ements with the same type. The arguments *sendbuf*,
count, and *datatype* define the send or input buffer and
recvbuf, *count* and *datatype* define the output buffer.
MPIREDUCE is called by all group members using the
same arguments for *count*, *datatype*, *op*, and *root*. All
processes in *comm* need to call this routine.

4.3 Sample of parallelized program for Poiseuille flow

*In this section, we explain how we parallelize the program for Poiseuille flow. The details we consider are kept as simple as possible to focus on the important features of MPI. In order to use MPI to explicitly pass messages between parallel processes, we must add message passing constructs to our program. To enable our programs to use MPI, we must include the MPI header file, **mpif.h**, in our source code **MPI.f**.*

```
1. PROGRAM MPI
2. INCLUDE 'poisflow.inc'
3. INCLUDE '/usr/include/mpif.h'
4. INTEGER ierror, NP, ID
5. CALL MPI_INIT(ierror)
6. CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NP, ierror)
7. CALL MPI_COMM_RANK(MPI_COMM_WORLD, ID, ierror)
8. CALL poisflow(ID, NP)
9. CALL datacollect(ID, NP)
10. CALL MPI_FINALIZE(ierror)
11. END
```

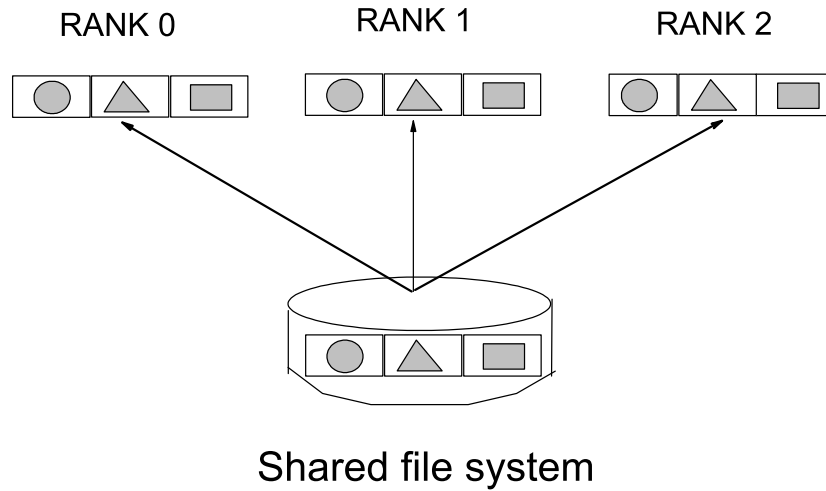


Figure 4.4: The input file on a shared file system.

Note that the program is executed in the SPMD (Single Program Multiple Data) model. All the nodes that run the program, therefore, need to see the same executable file with the same path name. The input file is located on a shared file system and each process reads data from the same file. The input file on a shared file system is shown in Fig. 4.4.

Line 1: *MPI program.*

Line 2: *Includes `poisflow.inc`, which defines parameters and global variables for the subroutine `poisflow` called in Line 8.*

Line 3: *Includes `mpif.h`, which defines MPI-related parameters such as `MPI_COMM_WORLD`. All Fortran procedures that use MPI subroutines have to include this file.*

-
- Line 4:** Declares *INTEGER* variables. **NP** is the number of processes, **ID** is the rank of the process and **ierror** is the Fortran return code.
- Line 5:** Calls the environmental subroutine, **MPI_INIT**, to initialize an MPI environment. **MPI_INIT** must be called once and only once before calling any other MPI subroutines.
- Line 6:** Calls the environmental subroutine, **MPI_COMM_SIZE**, to return the number of processes belonging to the communicator, **MPI_COMM_WORLD** specified in the first argument and defined in **mpif.h**. The communicator represents the group consisting of all the processes participating in the parallel job.
- Line 7:** Calls the environmental subroutine, **MPI_COMM_RANK**, to return the rank of the process within the communicator given as the first argument.
- Line 8:** Calls the subroutine **poisflow** performing the NEMD simulations for Poiseuille flow. The number of processes and the identifier of the current process are passed to the subroutine **poisflow** by two parameters. The program runs independently on different processes. In the subroutine **poisflow**, the initial velocities of all

*the particles are generated with the random number which relates to the **ID** of the process. This means the different processes run the same program with different initial random conditions for the momenta and avoids getting exactly the same results.*

Line 9: *Calls the subroutine **datacollect** to collect the data from all the processes. This subroutine will be analysed next.*

Line 10: *Calls **MPI_FINALIZE** to terminate MPI processing and no other MPI call can be made after this subroutine. Ordinary Fortran code can follow **MPI_FINALIZE**.*

Subroutine *datacollect* is called in *MPI.f* to collect the data from all the processes.

```
1.      SUBROUTINE datacollect(ID,NP)
2.
3.      INCLUDE 'poisflow.inc'
4.
5.      INCLUDE '/usr/include/mpif.h'
6.
7.      DOUBLE PRECISION temp(1:N)
8.
9.      INTEGER ID, NP, ierror, i
10.
11.     CALL MPI_REDUCE(data(1), temp(1), N, MPI_DOUBLE_PRECISION,
12.     &  MPI_SUM, 0, MPI_COMM_WORLD, ierror)
13.
14.     do 60 i=1, N
15.
16.         data(i)=temp(i)/NP
17.
18.     60  CONTINUE
19.
20.     if (ID.eq.0) then  CALL output
21.
22.     RETURN
23.
24.     END
```

Line 1: Subroutine *datacollect* which is called in *MPI* program.

- Line 2:** Includes **poisflow.inc**, which defines parameters and global variables for the subroutine **poisflow** called in MPI program.
- Line 3:** Includes **mpif.h**, which defines MPI-related parameters such as **MPI_COMM_WORLD**. All Fortran procedures that use MPI subroutines have to include this file.
- Line 4:** Declares **DOUBLE PRECISION** one dimensional array **temp** with the size, *N*.
- Line 5:** Declares **INTEGER** variables. **NP** is the number of processes, **ID** is the rank of the process and **ierror** is the Fortran return code.
- Line 6:** Calls the collective communication subroutine, **MPI_REDUCE** to apply a reduction operation, **MPI_SUM**, to the array **data** which is declared in **poisflow.inc** over the set of processes specified by the communicator, **MPI_COMM_WORLD**, and places the result in the array **temp** on root 0. Both the input and output buffers have the same number of elements, *N*, with the same data type specified as **MPI_DOUBLE_PRECISION**, which is one of the MPI data types. MPI provides several common operators by default, where **MPI_SUM** is one of them defined in

mpif.h. `MPI_REDUCE` is called by all group members using the same arguments. All processes in the communicator need to call this routine.

Line 7: Do loop over all the elements in the array **temp**.

Line 8: Calculates averages.

Line 9: Continues the loop.

Line 10: Checks if the process is **root**. If the process is **root**, write the data to **output** file, as the results are put in the array on root 0.

When an MPI program is written in the C language, there are some points which one should be aware of:

- Include **mpi.h** instead of **mpif.h**.
- C is case-sensitive. All of the MPI functions have the form **MPI_FUNCTION**, where MPI and the function name are in upper-case as in **MPI_INIT**. Constants defined in **mpi.h** are all in upper-case such as **MPI_INT**, **MPI_SUM**, **MPI_COMM_WORLD**.
- Those arguments of an MPI function call that specify the address of a buffer have to be given as pointers.
- Predefined MPI data types in C are different from Fortran bindings.

4.4 Running a parallelized program

In this section we explain how to compile, debug and run an MPI program.

Compiling and linking:

For Fortran, compile with a command like:

```
% f90 mpi.f -o mpi.exe -lmpi -lalan
```

For C, compile with a command similar to:

```
% cc mpi.c -o mpi.exe -lmpi -lalan
```

When compiling the MPI program, we must link to the MPI libraries, by using *-lmpi*. We must also link to the ELAN library, a communications library used for internode communication, by including *-alan* in our command line on the AlphaServer SC, the machine based at the APAC facilities at the Australia National University in Canberra, where most of our simulations were performed.

Running MPI jobs:

It is necessary to use the **prun** command to start an MPI executable, both when running within a batch job and when running small interactive test jobs. To run a small test with 4 processes (or tasks) where the MPI executable is called *mpi.exe*, enter the command:

```
% prun -n 4 mpi.exe
```

The argument to *-n* is the number of *mpi.exe* processes required to run. For larger jobs and production use, submit a job to the PBS batch system which allows access to all the processors on the machine. The batch file, *com-batch*, looks like:

```
# !/bin/csh
# PBS -P f15
# PBS -q normal
# PBS -l walltime=15:51:23, vmem=450MB, ncpus=4
# PBS -r n
# PBS -wd
prun -n 4 mpi.exe
```

Most jobs will require greater resources than are available to interactive processes. Larger jobs must be scheduled by the batch job system. We submit jobs to the portable batch system (PBS) specifying the project name, number of CPUs, the amount of memory, and the length of time needed. PBS runs the job when the resources are available, subject to constraints on maximum resource usage.

Options of note:

-P *f15*

The project, *f15*, which you want to charge the jobs resource usage to.

-q *normal*

The systems have a simple queue structure with three levels of priority, *normal*, *express* and *bonus*. The queue names reflect their priority.

-l *walltime* = 20 : 00 : 00

The wall clock time limit for the job. Time is expressed in seconds as an integer, or in the form:

hours : minutes : seconds.

-l *vmem* = 450MB

The total (virtual) memory limit for the job. It can be specified with units of “MB” or “GB” but only integer values can be given. A job will only run if there is sufficient free memory so making a sensible memory request will allow jobs to run sooner. A little trial and error may be required to find how much memory the jobs are using. *nqstat* lists jobs actual usage.

-l *ncpus* = 4

The number of CPUs required for the job to run on. The default is 1.

-r *n*

Specifies the job is not restartable, and if the job is executing on a node when it crashes, the job will not be requeued.

-wd

Start the job in the directory from which it is submitted. Normally jobs are started in the users home directory.

prun command is used to start an MPI executable, *mpi.exe*. By not specifying the *-n* option with the batch job, **prun** will start as many MPI processes as there have been CPUs requested with **qsub**. It is possible to specify the number of processes on the batch job **prun** command, as for the interactive case. To submit jobs to the queues, the simplest use of the **qsub** command is as follows

```
% qsub com-batch
```

Debugging a program that executes in parallel can be quite difficult. If the program crashes, for instance, each process will create a separate core file, in its own directory, with a name like *coredir.1*. **dbx** debugger can be used to query each core file by itself. If we can assume that the executable program is called *mpi.exe*, and that we want to look at the core file associated with process 4, and we just want to get a traceback of “where” process 4 was when it crashed, it would be enough to type:

```
dbx mpi.exe coredir.4/core  
  
where  
  
quit
```

Command **where** is used to get a traceback of the point where the process crashed.